山东大学
SHANDONG UNIVERSITY

崇新学堂

2024 — 2025 学年第一学期

实验报告

课程名称：Introduction to EECS Lab

实验名称：DL14 - I'm the Map!

学生姓名：胡君安、陈焕斌、黄颢

实验时间：2024 年12月25日

DL14 - I'm the Map!

1. Introduction

The goal of this experiment is to enable robots to complete navigation and obstacle avoidance tasks by dynamically generating environmental maps. The main contents of the experiment include:

1. Build a dynamic map generator to process reliable and unreliable sensor data.

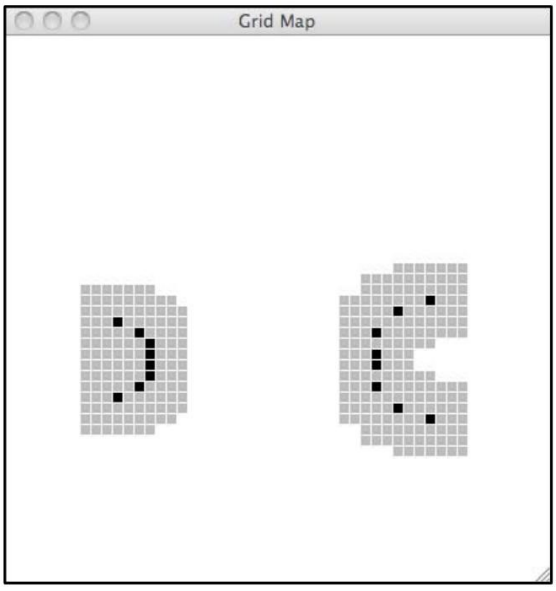
2. Implement robust map generation based on Bayesian state estimation.

3. Integrate the map generation and planning module to achieve real-time path planning and re-planning.

4. Test and optimize the system in simulated and real robot environments.

2. Experimental step

Step 1: Sensor Input



Check Yourself 1

We have defined a SensorInput class for testing in idle that simulates the io.SensorInput class in soar. Consider these two possible sensor input instances (each has a list of 8 real-valued sonar readings and a pose). testData = [SensorInput([0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2], util.Pose(1.0, 2.0, 0.0)), SensorInput([0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4],

util.Pose(4.0, 2.0, -math.pi))].Be sure you understand why they give rise to the map shown below. Remember that the black squares are the only ones that are marked as occupied as a result of the sonar readings; the gray squares are the places that the robot cannot occupy (because it would collide with one of the black locations)

Each SensorInput object contains:

1.A list of ultrasonic sensor readings (8 readings in this example).

[0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2] indicates that the readings of all 8 sensors are 0.2 meters.

[0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4] indicates that the readings of all 8 sensors are 0.4 meters.

Each reading represents the distance to the obstacle detected by the sensor.

2.The pose of the robot (util. Pose type), including:

X and y coordinates (the position of the robot in the world coordinates).

Theta (direction angle): The angle (in radians) of the robot's orientation.

The first set of data: Obstacles should surround (1.0, 2.0), with a distance of 0.2 meters.

The second set of data: Obstacles should surround (4.0, 2.0), with a distance of 0.4 meters.

Step 2: Implement the MapMaker Class

Implement the MapMaker class. It will be called as follows: MapMaker(xMin, xMax, yMin, yMax, gridSquareSize) Remember to initialize the startState attribute and to define a getNextValues method that marks the cells at the end of the sonars rays in the input.

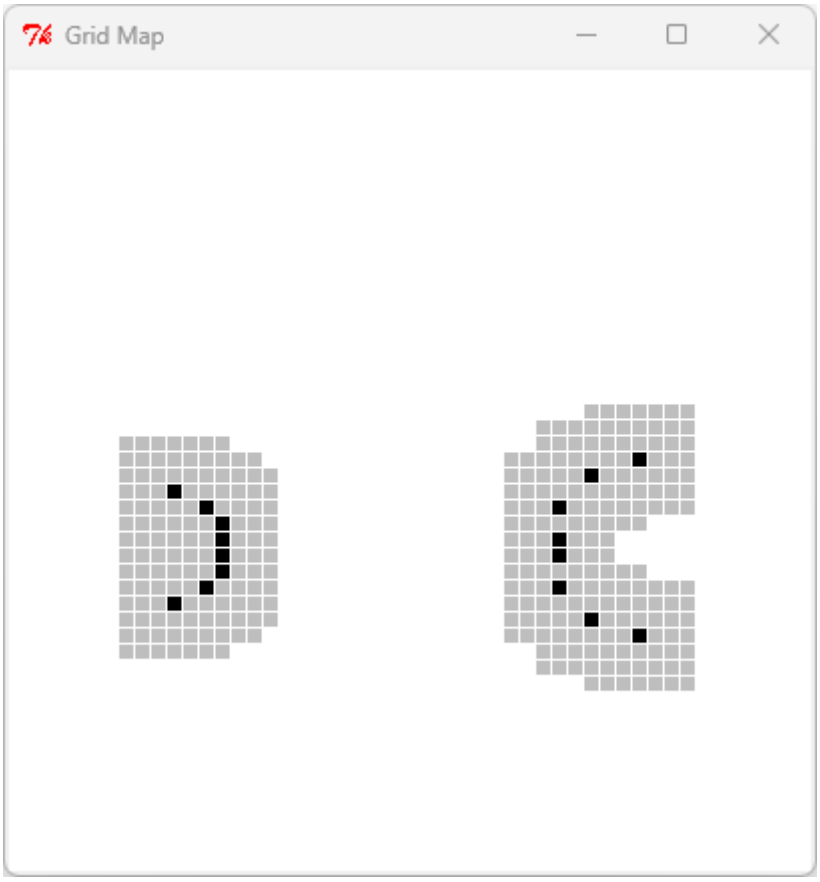
```
Python
class MapMaker(sm.SM):
    def __init__(self, xMin, xMax, yMin, yMax, gridSquareSize):
        self.startState = dynamicGridMap.DynamicGridMap(xMin, xMax, yMin, yMax,
gridSquareSize)

    def getNextValues(self, state, inp):
        sonarReadings = inp.sonars
        for idx, distance in enumerate(sonarReadings):
            if distance < sonarDist.sonarMax:
                detectedPoint = sonarDist.sonarHit(distance, sonarDist.sonarPoses[idx],
inp.odometry)
                detectedGrid = state.pointToIndices(detectedPoint)
                originPoint = sonarDist.sonarHit(0, sonarDist.sonarPoses[idx],
inp.odometry)
                originGrid = state.pointToIndices(originPoint)
                if not state.occupied(detectedGrid):
                    state.setCell(detectedGrid)
        return state, state
```

We initialized a dynamic grid map DynamicGridMap as the initial state of the system and configured the map based on the given range and grid size. In the getNextValues method, we

traverse each reading of the sensor to determine if it is less than the maximum effective distance. If it is valid, we calculate the world coordinates of the end and beginning points of the sonar ray and convert them into integer grid indexes. Then, we use these indexes to generate all grid points on the ray path, clean up the points on the path by calling the clearCell method, and call the setCell method at the end to mark obstacles. This method ensures that the area on the path is marked as passable and the position of the obstacle is correctly mapped to the map.

Step 3: Test the MapMaker Class

Test your map maker inside idle (be sure to start with the -n flag). by doing this: testMapMaker(testData) It will make an instance of your MapMaker class, and call transduce on it with the testData from the Check Yourself question. Verify that your results match those in the figure.



The above is the code test result, which shows that it conforms to the diagram given by the experimental handout.

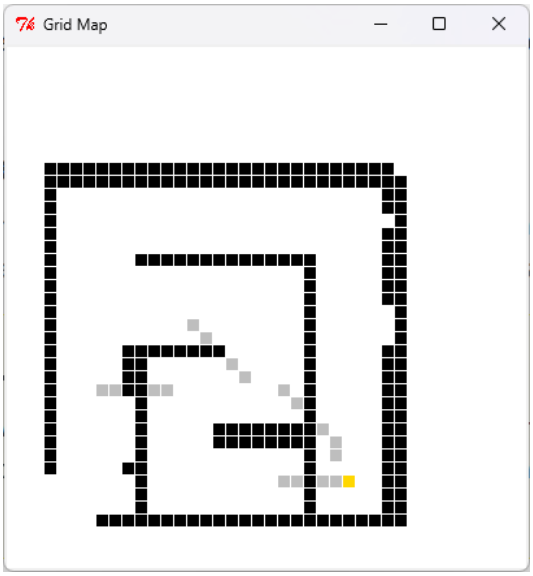
Step 4: Test the Code in Soar

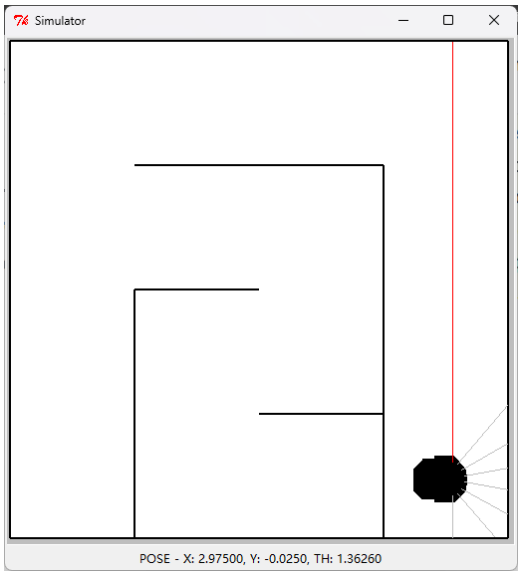
Now, test your code in soar. The file mapAndReplanBrain.py contains the necessary state machine combinations to connect all the parts of the system together into a brain that can run in soar.

Checkoff 1.

Wk.14.2.1:Show the map that your mapmaker builds to a staff member. If it does anything surprising, explain why. How does the dynamically updated map interact with the planning and replanning process?

The dynamically updated map records the location of obstacles detected by robot sensors in real time, marks this information on the map, and dynamically adjusts unknown areas to known states, providing the latest environmental data for path planning. When the robot is executing the initial path, if it finds that some grids in the path are impassable due to detected obstacles, the system will trigger a re-planning mechanism to recalculate the new path to the target point based on the current map, ensuring that the robot can flexibly bypass obstacles and continue to move forward, achieving dynamic adaptation to environmental planning and navigation.





Step 5: Noise Dilemma

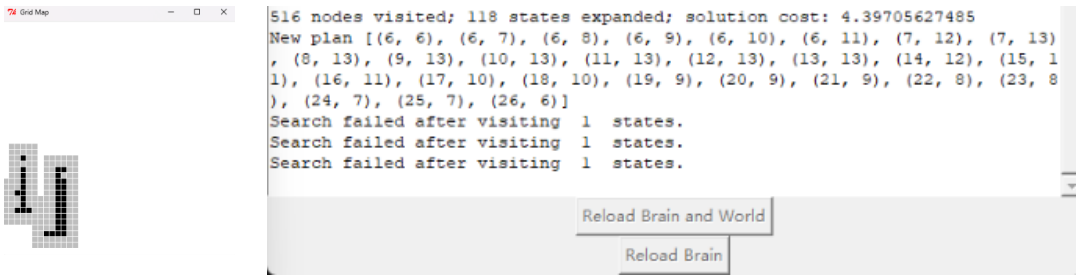
By default, the sonarreadings in soar are perfect. But the sonarreadings in a real robot are nothing like perfect. Find the line in mapAndReplanBrain.py that says:

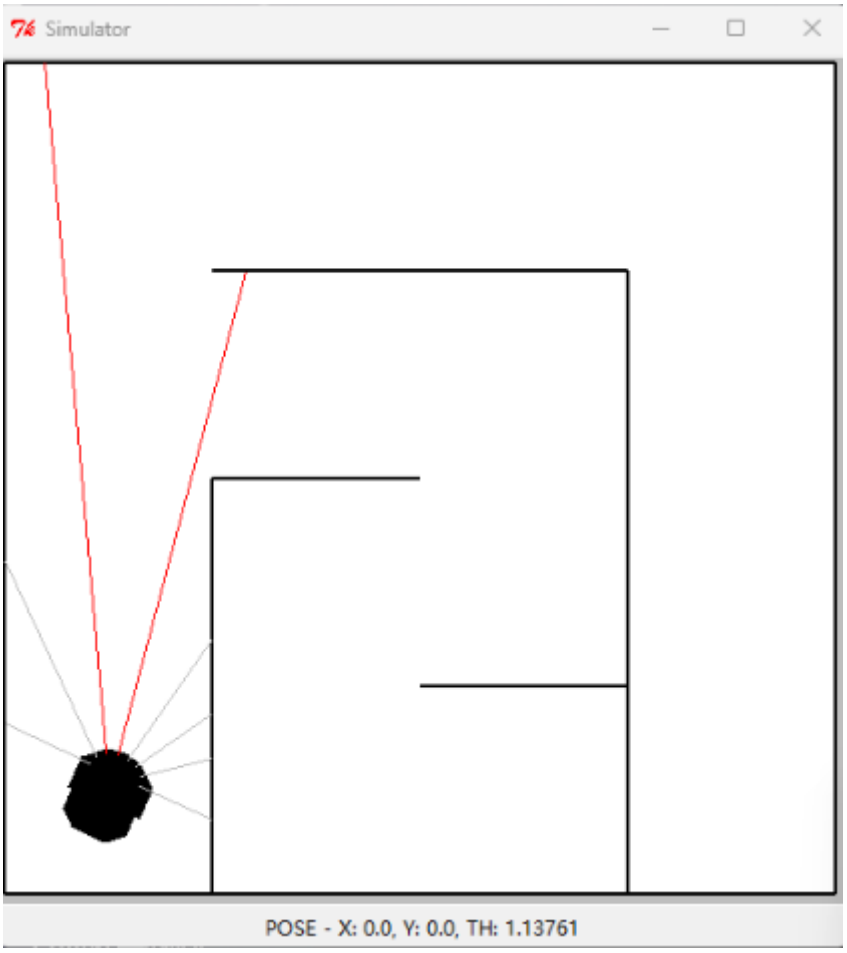
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: noNoise

and change it to one of

soar.outputs.simulator.SONAR_VARIANCE = lambda mean: smallNoise
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: mediumNoise.

This increases the default variance (width of gaussian distribution) for the sonar noise model to a non-zero value.





Check Yourself 2

Run the brain again in these noisier worlds. Why doesn't it work? How does the noise in the sensor readings affect its performance?

From the above experimental results, it can be seen that when noise is introduced, the robot may incorrectly mark the grid, such as mistakenly marking blank areas as obstacles, or failing to correctly detect actual obstacles, which will lead to incorrect maps in the path planning process. In addition, noise may also cause the robot to think that the target area is blocked by obstacles, or there are impassable grids on the path, triggering repeated ineffective re-planning, and even causing the robot to fall into a dead loop or unable to find a feasible path, resulting in the inability to complete the entire journey.

Step 6: Improve the Code

Improve your MapMaker class to take advantage of this information. You will probably find the procedure util.lineIndices(start, end) useful: start and end should each be a pair of (x, y) integer grid cell indices; the return value is a list of (x, y) integer grid cell index pairs that constitute the line segment between start and end (including both start and end). You can think of these cells as the set of grid locations that could reasonably be marked as being clear, based on a sonar measurement. Be sure not to clear the very last point, which is the one that you are already marking as occupied; although it might work now, if you clear and then mark that cell each time, it will cause problems in Section 4, when we use a state estimator to aggregate the evidence we get about each grid cell over time.

```
Python
class MapMaker(sm.SM):
    def __init__(self, xMin, xMax, yMin, yMax, gridSquareSize):
        self.startState = dynamicGridMap.DynamicGridMap(xMin, xMax, yMin, yMax,
gridSquareSize)

    def getNextValues(self, state, inp):
        sonarReadings = inp.sonars
        for idx, distance in enumerate(sonarReadings):
            if distance < sonarDist.sonarMax:
                detectedPoint = sonarDist.sonarHit(distance, sonarDist.sonarPoses[idx],
inp.odometry)
                detectedGrid = state.pointToIndices(detectedPoint)
                originPoint = sonarDist.sonarHit(0, sonarDist.sonarPoses[idx],
inp.odometry)
                originGrid = state.pointToIndices(originPoint)
                if not state.occupied(detectedGrid):
                    state.setCell(detectedGrid)
                indices = util.lineIndices(originGrid, detectedGrid)
                indices1 = indices[: -1]
                for i in indices1:
                    state.clearCell(i)
        return state, state
```

By calling the util.lineIndices function, we can calculate all the mesh elements between the sonar starting point (originGrid) and the detected obstacle position (detectedGrid). The role of the lineIndices function is to find the index of each mesh that a straight line passes through from the starting point to the target point. These mesh positions may include the sonar starting point, detection point, and all intermediate mesh elements where this line intersects with other meshes. The indices returned by this function are a list containing all the mesh indexes that the straight line passes through. ClearCell (i) will mark the grid cells at the specified index position as unoccupied, usually because the grid is considered a blank area or the position is mistakenly identified as an obstacle and needs to be cleared during sonar data updates. By clearing these intermediate grid cells, it avoids mistakenly labeling these areas that should have been blank as obstacles and maintains the accuracy of the map.

Step 7: Test the New MapMaker

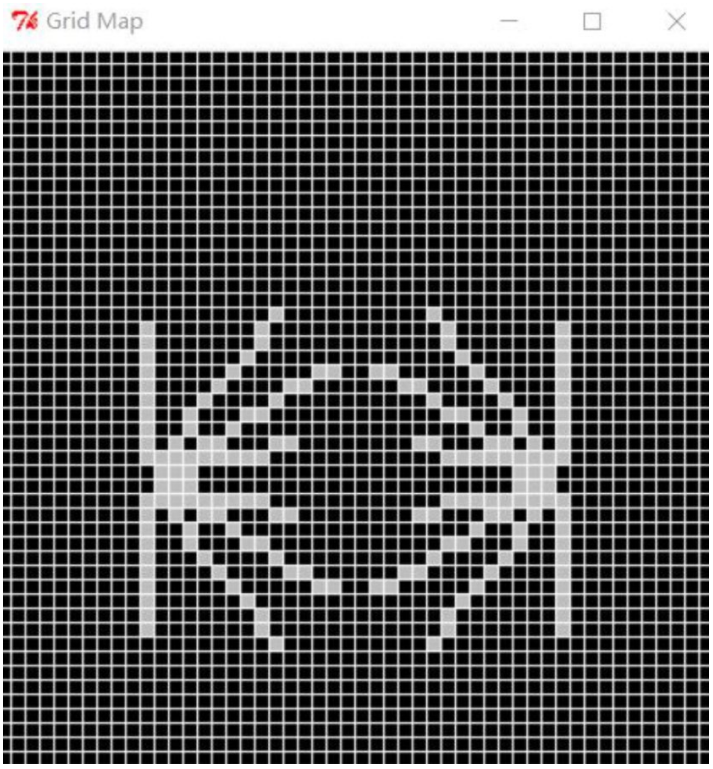
Test your new MapMaker in Idle by doing testMapMakerClear(testClearData) Note that this is testMapMakerClear, which is a different procedure from testMapMaker. It will create an instance of your map maker and set all of the grid squares to be occupied, initially. Then it will call transduce with this input: testClearData = [SensorInput([1.0, 5.0, 5.0, 1.0, 1.0, 5.0,

5.0, 1.0], util.Pose(1.0, 2.0, 0.0)), SensorInput([1.0, 5.0, 5.0, 1.0, 1.0, 5.0, 5.0, 1.0], util.Pose(4.0, 2.0, -math.pi))]

Check Yourself 3

Predict what the resulting map should look like, and make sure your code produces the right thing.

The generated map will display the obstacle detection results of the robot at different positions. The first SensorInput object indicates that the robot is located at (1.0, 2.0), facing the positive X-axis, and the distance measured by sonar is 1.0 and 5.0 meters, respectively. Due to this distance information, the obstacles in the map will be presented at positions closer to the robot (1.0 meters) and farther away (5.0 meters). The layout of the obstacles will show a certain arc distribution, because the distance of obstacles measured by sonar sensors in different directions is different. The second SensorInput object indicates that the robot is located at (4.0, 2.0), facing the negative X-axis. The sonar data is the same as the first one, so obstacles will appear closer to this position, while also displaying obstacles that are farther away. Overall, the generated two-dimensional map will depict the distribution of obstacles based on these measurement data, including obstacles that are close and obstacles that are far away. The obstacles in the map will show two different positions and directions. Combined with the analysis and running results, we obtained the following images:



Step 8: Run It in Soar

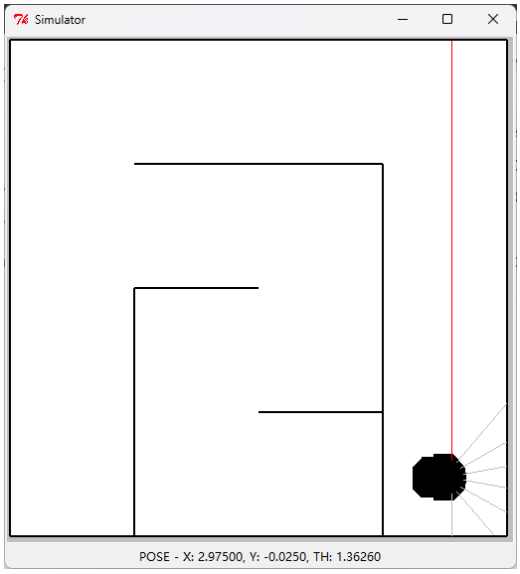
Run mapAndReplanBrain in soar again and make sure you understand what happens with

both no noise and medium noise.

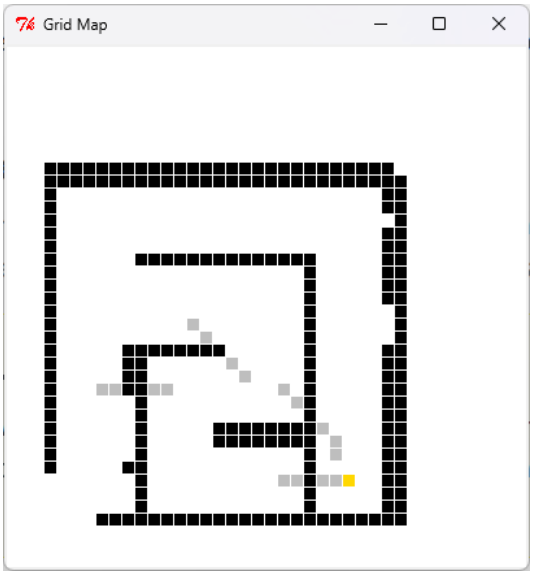
Checkoff 2.

Wk.14.2.2: Show your new map maker running, first with no noise and then with medium noise. We don’t necessarily expect it to work reliably: but you should explain what it’s doing and why.

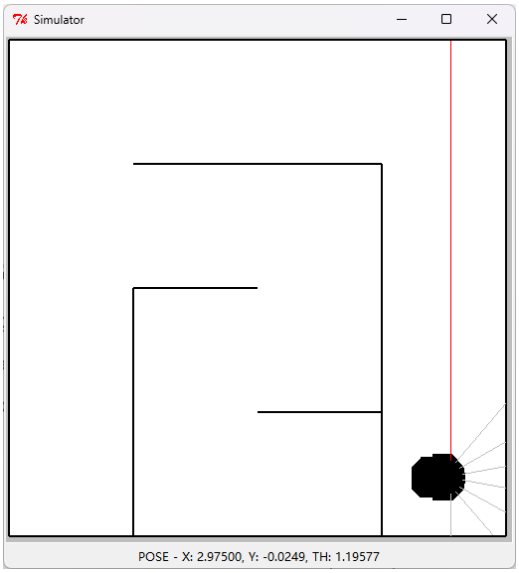
We can know that in the case of smallNoise, our improved code has a significant function of eliminating misjudgments. However, in the case of mediumNoise, the power of our improved code is not enough. Due to too much noise, frequent recognition errors occur and cannot be corrected smoothly.



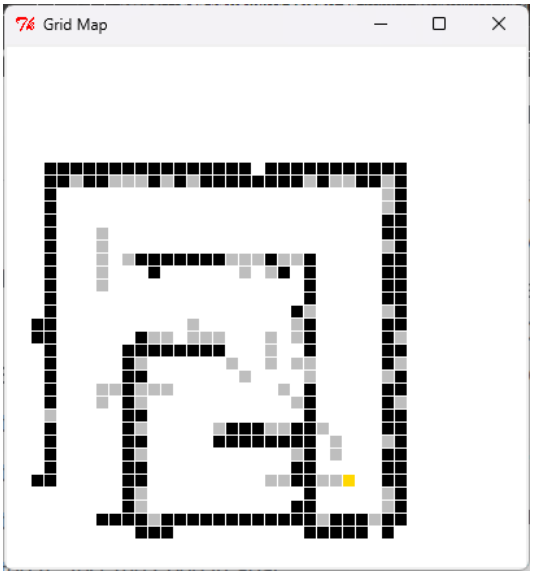
noNoise



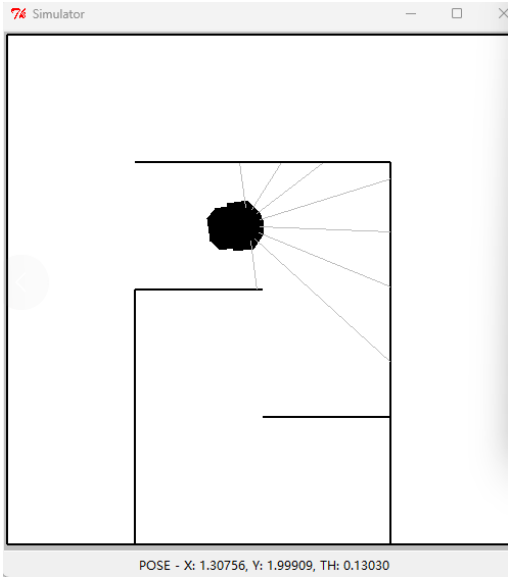
noNoise



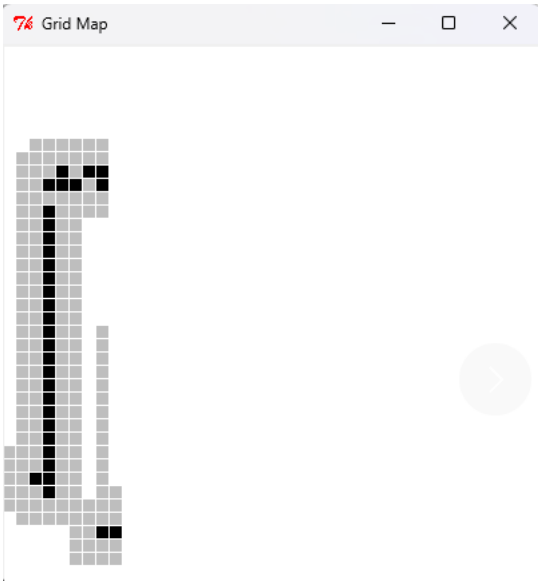
smallNoise



smallNoise



mediumNoise



mediumNoise

Check Yourself 4

Remember that the sonar beams can sometimes bounce off of obstacles and not return to the sensor, and that when we say a square is clear, we say that it has nothing anywhere in it. What do you think the likelihood is that we observe a cell to be free when it is really occupied? That we observe it as a hit when it is really not occupied? What should the prior

(starting) probabilities be that any particular cell is occupied? Decide on possible values for the state of the cell. Assume that the observation can be either 'hit', if there is a sonar hit in the cell or 'free' if the sonar passes through the cell. To forestall confusion, pick names for the internal states that are neither 'hit' nor 'free'. If you are having trouble formulating the starting distribution, observation and transition models for the state estimator, talk to a staff member.

We can assume that the probability of correct recognition is 0.8, the initial probability of any specific cell being occupied is set to 0.5, and we can use this probability distribution to proceed to the next step.

Step 9: Create an Instance of ssm.StochasticSM

Write code in bayesMapSkeleton.py to create an instance of ssm.StochasticSM that models the behavior of a single grid cell.

```
Python
def oGivenS(s):
    if s == 'occupied':
        return dist.DDist({'hit':0.8, 'free':0.2})
    else:
        return dist.DDist({'hit':0.2, 'free':0.8})

def uGivenAS(a):
    def GivenS(s):
        if s == 'occupied':
            return dist.DDist({'occupied':0.8, 'not occupied':0.2})
        else:
            return dist.DDist({'occupied':0.2, 'not occupied':0.8})
    return GivenS

startDistribution = dist.DDist({'occupied':0.5, 'notOccupied':0.5})
cellSSM = ssm.StochasticSM(startDistribution, uGivenAS, oGivenS)
```

We combined some of the statistical models defined in Check Yourself 4 to create a dynamic system that describes the behavior of individual grid cells through the StochasticSM class.The above probabilities are all based on assumptions.

Step 10: Test the Grid Cell Model

Test your grid cell model by doing

testCellDynamics(cellSSM, yourTestInput)

where cellSSM is an instance of ssm.StochasticSM and yourTestInput is one of the lists below. It will create an instance of a state estimator for a single grid cell and feed it a stream of observations. Then it will call transduce with the data input.

The results are as follows.

```
>>> testCellDynamics(cellSSM, mostlyFree)
[DDist(occupied: 0.320000, not occupied: 0.680000), DDist(occupied: 0.263158, not occupied: 0.736842), DDist(occupied: 0.249180, not occupied: 0.750820), DDist(occupied: 0.542214, not occupied: 0.457786)]
>>> testCellDynamics(cellSSM, mostlyHits)
[DDist(occupied: 0.680000, not occupied: 0.320000), DDist(occupied: 0.736842, not occupied: 0.263158), DDist(occupied: 0.750820, not occupied: 0.249180), DDist(occupied: 0.457786, not occupied: 0.542214)]
```

Step 11: About BayesGridMap

Now it is time to think through a strategy for implementing the BayesGridMap class; don't start implementing just yet You will have to manage the initialization and state update of the state estimator machines in each cell yourself. You should be sure to call the start method on each of the state-estimator state machines just after you create this grid. You will also, whenever you get evidence about the state of a cell, have to call the step method of the estimator, with the input (o, a), where o is an observation and a is an action; we will be, effectively, ignoring the action parameter in this model, so you can simply pass in None for a.

Step 11 provides the idea of how to implement the BayesGridMap class. First, we need to manage the state estimator (StateEstimator) of each cell in the grid. Each cell should have a state estimator responsible for updating and storing the state of the cell. We need to ensure that when creating the grid, we immediately call the start method on the state estimator of each cell to initialize their state. After obtaining observations about the cell (such as sonar measurements), you need to call the step method of the cell state estimator, passing in observations and actions as inputs. In this model, action (a) will be ignored, so None can be passed in as an action.

Step 12: Object Instances

Solve this tutor problem Wk.14.2.3 on making collections of object instances.

Part 1:

Consider the following code:

```
def lotsOfClass(n, v):
   one = MyClass(v)
   result = []
   for i in range(n):
       result.append(one)
   return result

class10 = lotsOfClass(10, 'oh')

class10[0].v = 'no'
```

1. What is the value of `class10[0].v`: 'no'
2. What is the value of `class10[3].v`: 'no'

Part 2:

```
Python
def lotsOfClass(n, v):
    r = [MyClass(v) for _ in range(n)]
    return r
```

Part 3:

```
Python
def lotsOfClass(n, v):
    def initFunction(n):
        return MyClass(v)
    return util.makevectorFill(n, initFunction)
```

InitFunction is a closure that only accepts the parameter n, but does not actually use it. The task of initFunction is to create a new instance of MyClass (v) every time it is called. It uses the variable v of the outer scope, which is a typical feature of closures.

Step 13: Some Useful Methods

Here is some further description of the methods you'll need to write. Remember that the grid of values in a DynamicGridMap is stored in the attribute grid. We don't need to write the init method, because it will be inherited from DynamicGridMap.

The core task of this step is to write a set of methods for operating dynamic grid maps based on the DynamicGridMap class. These methods involve initializing grid maps, updating cell states, calculating occupancy probabilities, and their application in path planning.

1. **makeStartingGrid:**

Create a two-dimensional array (a list of lists), where each element is an instance of seFast. StateEstimator. The size of the grid is determined by the xN and yN properties of the current object. Use the util.make2DArrayFill method to achieve array filling.

2. **setCell :**

Receive a tuple parameter containing the x and y indexes, update the fine-state machine of the specified cell to indicate that the cell was hit by sonar detection. At the same time, repaint the cell to reflect its color change.

3. **clearCell:**

Receive a tuple parameter containing x and y indexes, update the finite-state machine of the specified cell to indicate that the cell is idle. At the same time, redraw the cell to reflect its color change.

4. **occProb:**

Returns the occupancy probability of the specified cell, which is a floating-point number between 0 and 1 used to display the transparency of the cell color (implemented through the squareColor method).

5. **occupied:**

Return whether the specified cell should be considered occupied (for path planning). The basis for determining the occupied status is the probability of the cell being occupied (the return value of the occProb method), and an appropriate threshold needs to be selected for experimentation to determine the judgment criterion.

Step 14: Implement the BayesGridMap Class

Now, implement the BayesGridMap class in bayesMapSkeleton.py. It already has the squareColor method defined.

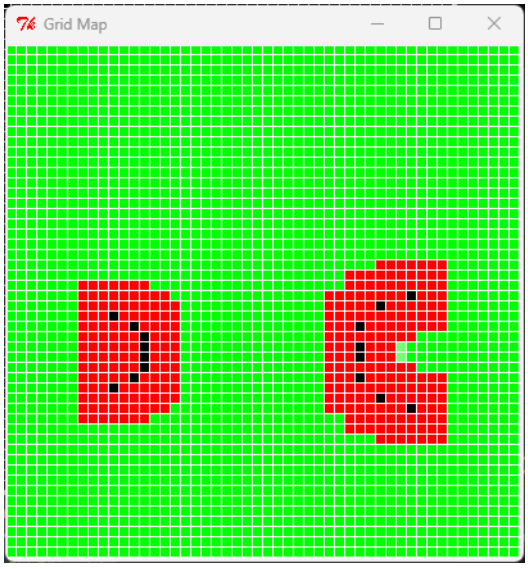
```
Python
class BayesGridMap(dynamicGridMap.DynamicGridMap):

    def squareColor(self, (xIndex, yIndex)):
        p = self.occProb((xIndex, yIndex))
        if self.robotCanOccupy((xIndex,yIndex)):
            return colors.probToMapColor(p, colors.greenHue)
        elif self.occupied((xIndex, yIndex)):
            return 'black'
        else:
            return 'red'

    def occProb(self, (xIndex, yIndex)):
        return self.grid[xIndex][yIndex].state.prob('occupied')

    def makeStartingGrid(self):
        r = util.make2DArrayFill(self.xN, self.yN, lambda x, y:
seFast.StateEstimator(cellSSM))
        for x in range(self.xN):
            for y in range(self.yN):
                r[x][y].start()
        return r

    def setCell(self, (xIndex, yIndex)):
        self.grid[xIndex][yIndex].step(('hit', None))
        self.drawSquare((xIndex, yIndex))

    def clearCell(self, (xIndex, yIndex)):
        print 'xIndex:', xIndex, 'yIndex:', yIndex
        self.grid[xIndex][yIndex].step(('free', None))
        self.drawSquare((xIndex, yIndex))

    def occupied(self, (xIndex, yIndex)):
        if self.occProb((xIndex, yIndex)) > 0.5:
            return True
        return False
```

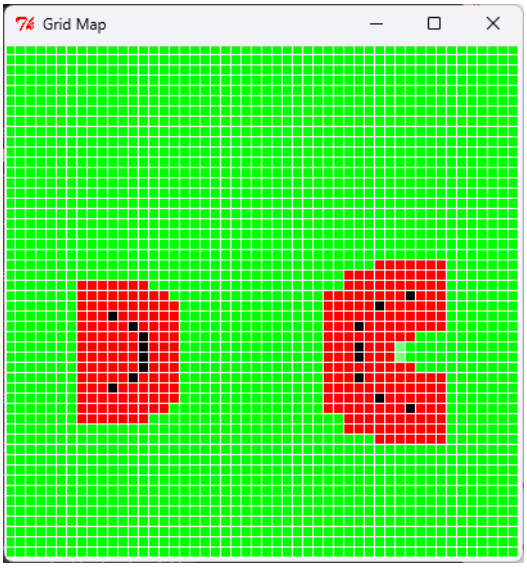
Step 15: Test the Code in Idle

Test your code in Idle by:

• Changing your MapMaker to use bayesMap.BayesGridMap instead of dynamicGridMap.DynamicGridMap. No further change to that class should be necessary.

• Running mapMakerSkeleton.py in Idle, and then typing in the shell:

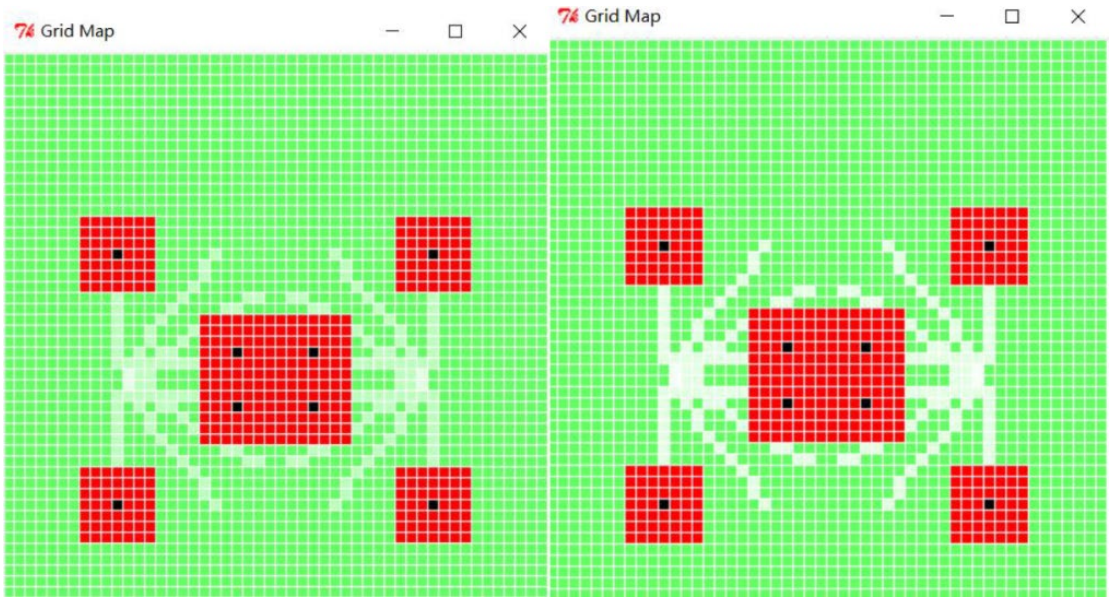




Check Yourself 5

Try it with two updates. Try it with testClearData. Be sure it all makes sense.

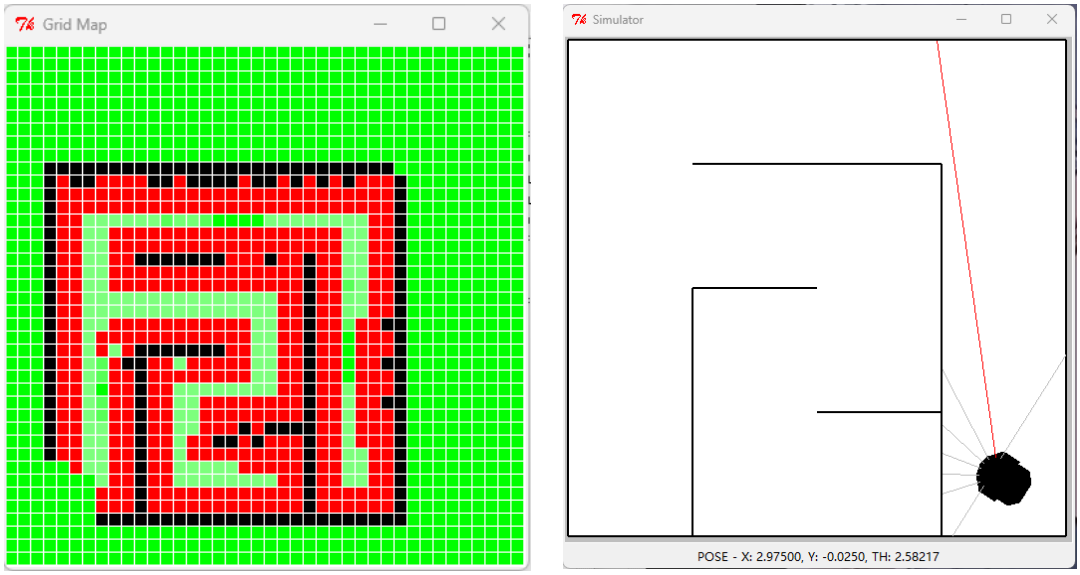
The generated Grid Map is shown in the following figure.



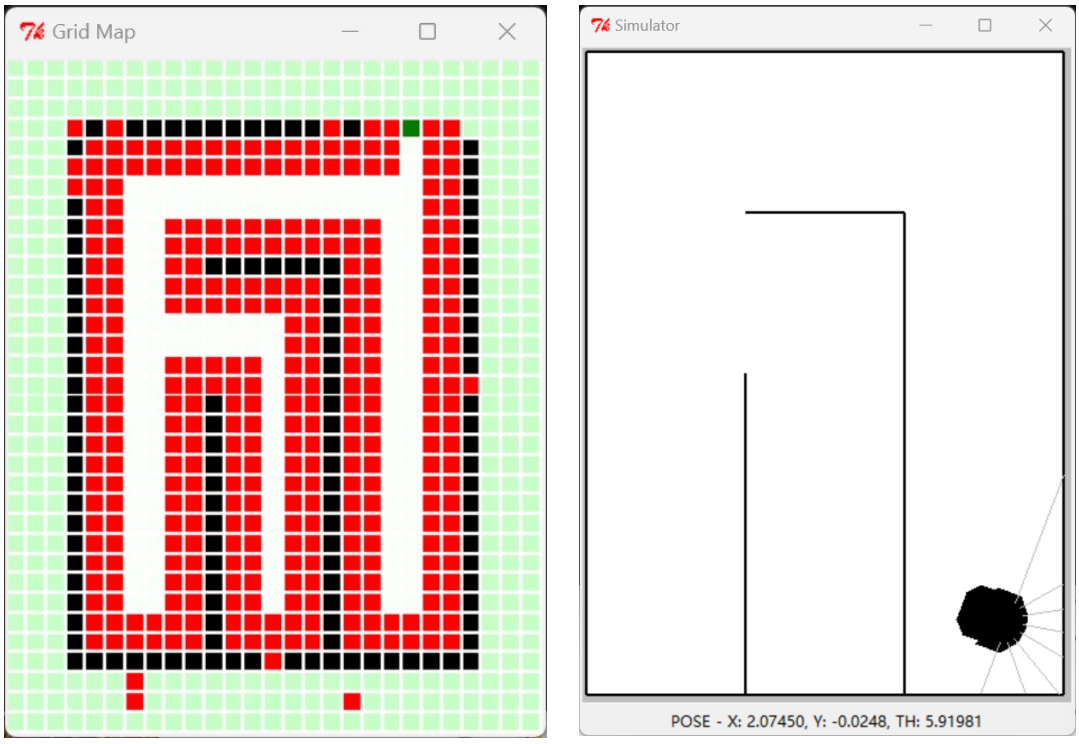
Combining the above diagram, we can see that the effect after two map updates is significantly better than that after only one update.

Step 16: Test the Mapper in Soar

Now, test your mapper in soar, by running mapAndReplanBrain as before. You might find it particularly useful to use the step button.



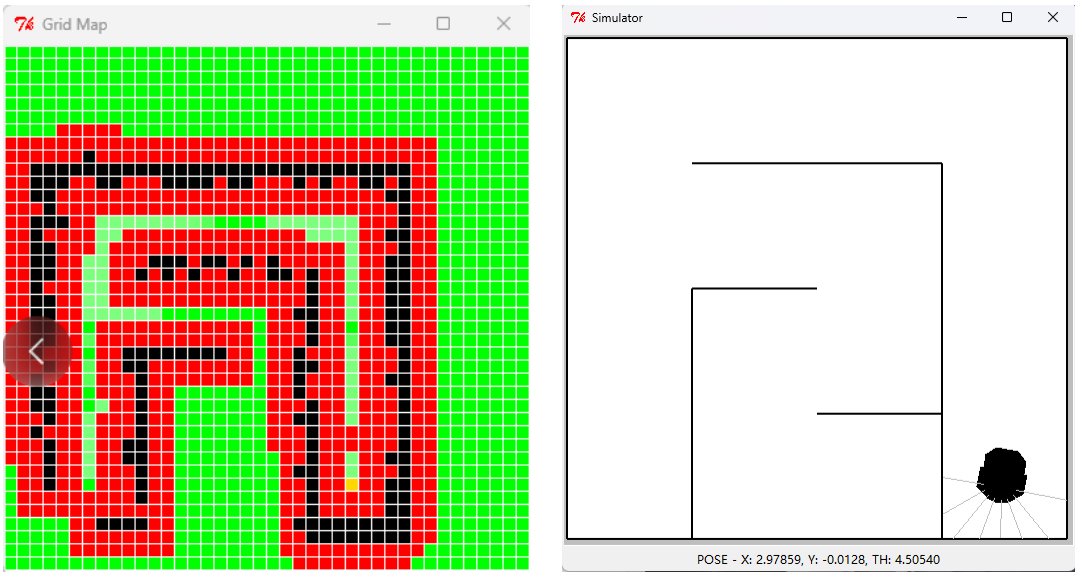
The car can complete the entire journey normally.



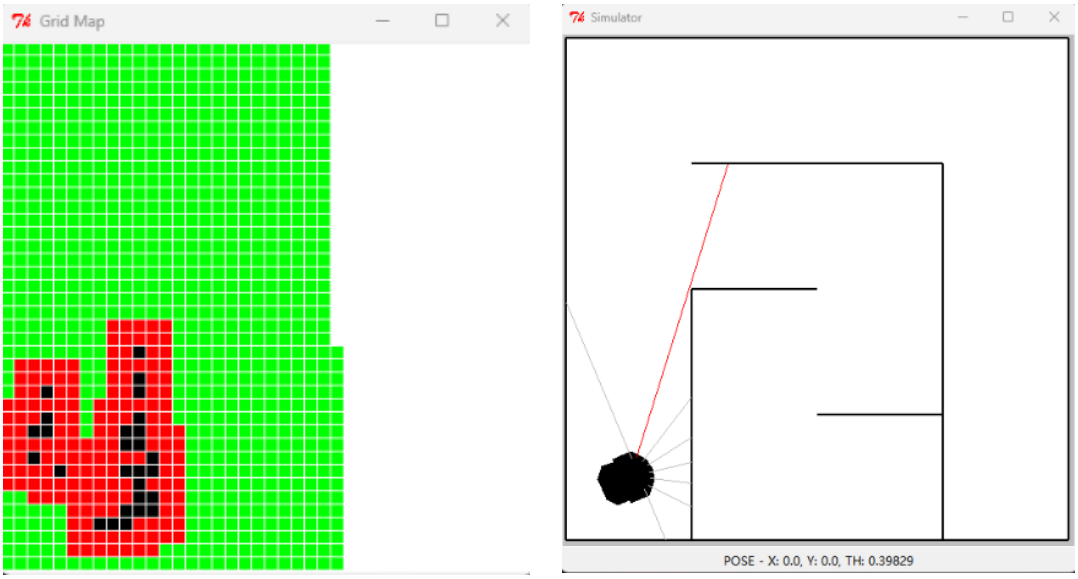
Checkoff 3.

Wk.14.2.4: Demonstrate your mapper in mapAndReplanBrain using your BayesMap module with medium noise and high noise. If it doesn't work with high noise, explain what the issues are, but you don't necessarily need to have it working with high noise.

mediumNoise:



bigNoise:




```
Search failed after visiting  1  states.
```

According to the simulation results, it can be seen that the robot can still reach the endpoint under medium noise conditions, but due to its incorrect recognition of road conditions at the fork in the road, it did not enter the starting fork at all.

In the case of bigNoise, the robot is unable to reach the finish line due to too many recognition errors, sometimes even displaying errors right after starting the robot, and sometimes my soap crashes, which may be related to my own computer.

Although the new method has advantages, it will accumulate the number of incorrect recognitions in the case of bigNoise, making it difficult for subsequent robot recognitions to

continue.

Checkoff 4.

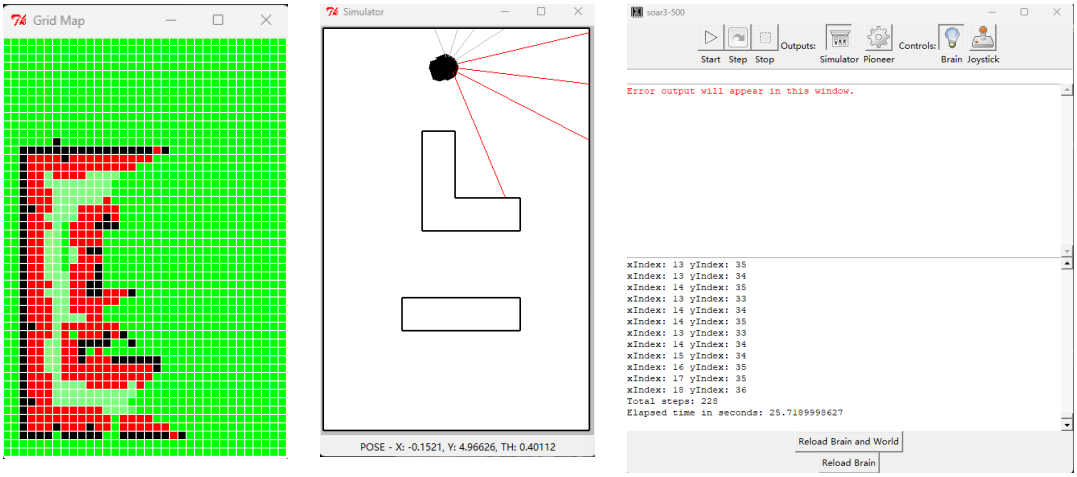
Wk.14.2.5: Demonstrate your mapper on a real robot. You can move the obstacles around in the playpen for added fun, but be sure that you don't make it impossible to go from the start to the goal.

To be improved.

Step 17: Speed Racer

Check Yourself 6

Run your robot through raceWorld and see what score you get. Write this down, because it's your baseline for improvement.



Total Steps: 228

Elapsed time in seconds: 25.7189998627

Step 18: Go Faster

Implement some improvements to make the robot go faster.

Check Yourself 7

Post your best scores in simulation on raceWorld and lizWorld on the board.

The best score is 6.7s !!! An unbelievable score!

We choose to change the gain of the motion controller to make the robot go faster.

```
Python
# Parameters in motion controller
move.MoveToDynamicPoint.forwardGain = 2
move.MoveToDynamicPoint.rotationGain = 2
move.MoveToDynamicPoint.angleEps = 0.25
```

We also optimized the **class** ReplannerWithDynamicMap(sm.SM).

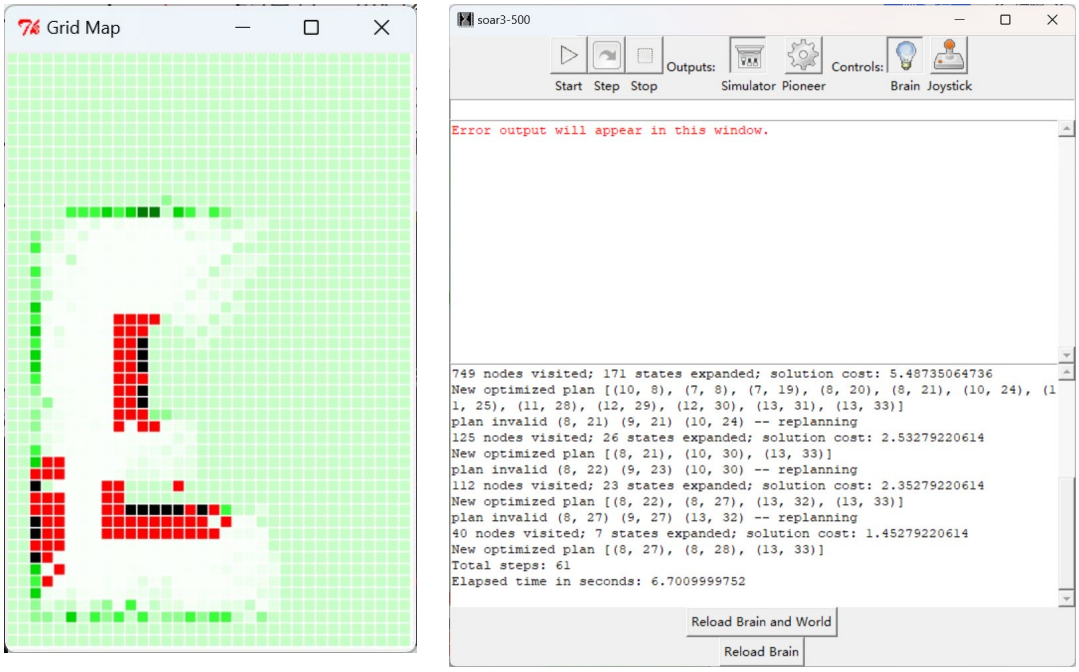
```
Python
class ReplannerWithDynamicMap(sm.SM):
    def __init__(self, goalPoint):
        self.goalPoint = goalPoint
        self.startState = None

    def optimizePath(self, map, indices):
        if not indices or len(indices) < 3:
            return indices

        optimized = [indices[0]]
        i = 0

        while i < len(indices)-2:
            p1 = indices[i]
            p2 = indices[i+1]
            p3 = indices[i+2]

            dx1 = p2[0] - p1[0]
            dy1 = p2[1] - p1[1]
            dx2 = p3[0] - p2[0]
            dy2 = p3[1] - p2[1]

            if dx1 * dy2 != dx2 * dy1:
                optimized.append(p2)
                i += 1
            else:
                i += 2

        optimized.append(indices[-1])
        return optimized

    def getNextValues(self, state, inp):
        (map, sensors) = inp
        dynamicsModel = gridDynamics.GridDynamics(map)
        currentIndices = map.pointToIndices(sensors.odometry.point())
        goalIndices = map.pointToIndices(self.goalPoint)

        if timeToReplan(state, currentIndices, map, goalIndices):
            def h(s):
                return self.goalPoint.distance(map.indicesToPoint(s))
            def g(s):
                return s == goalIndices

            plan = ucSearch.smSearch(dynamicsModel, currentIndices, g,
                                     heuristic = h, maxNodes = 5000)

            if state:
                map.undrawPath(state)

            if plan:
                pathIndices = [s[:2] for (a, s) in plan]
                state = self.optimizePath(map, pathIndices)
                print 'New optimized plan', state
                map.drawPath(state)
            else:
```

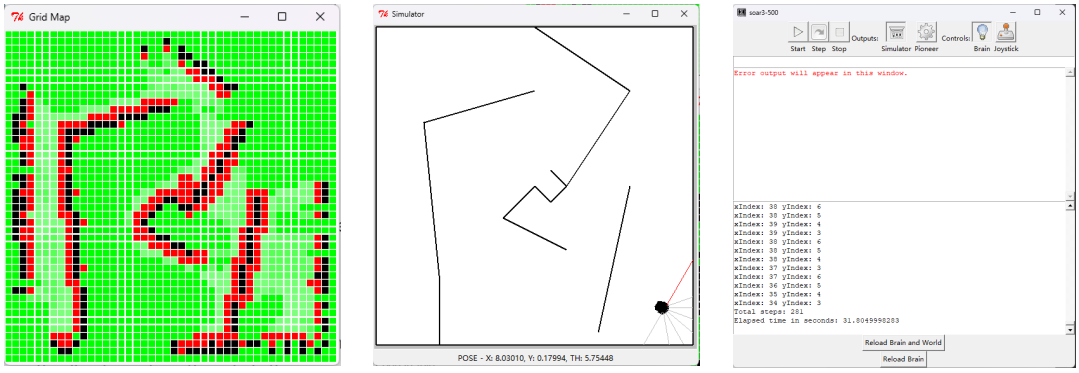
```
                map.drawPath([currentIndices, goalIndices])
                state = None

        if not state or (currentIndices == state[0] and len(state) == 1):
            return (state, sensors.odometry)
        elif currentIndices == state[0] and len(state) > 1:
            map.drawSquare(state[0])
            state = state[1:]
            map.drawPath(state)
        return (state, map.indicesToPoint(state[0]))

def timeToReplan(plan, currentIndices, map, goalIndices):
    return plan == None or planInvalidInMap(map, plan, currentIndices) or \
            (plan == [] and not goalIndices == currentIndices)

def planInvalidInMap(map, plan, currentIndices):
    if len(plan) == 0:
        return False
    wayPoint = plan[0]
    for p in util.lineIndicesConservative(currentIndices, wayPoint)[1:]:
        if not map.robotCanOccupy(p):
            print 'plan invalid', currentIndices, p, wayPoint, '-- replanning'
            return True
    return False
```

1. Added optimizePath() method that:
    - Takes path indices and removes redundant waypoints
    - Keeps only points where direction changes
    - Uses slope comparison to detect direction changes
    - Always preserves start and end points
2. Modified getNextValues() to:
    - Extract path indices from plan
    - Call optimizePath() to optimize the path
    - Use optimized paths for execution and visualization

This optimization will:

- Reduce the number of stopping points
- Allow longer straight-line movements
- Minimize unnecessary rotations
- Keep the path safe by preserving critical turning points

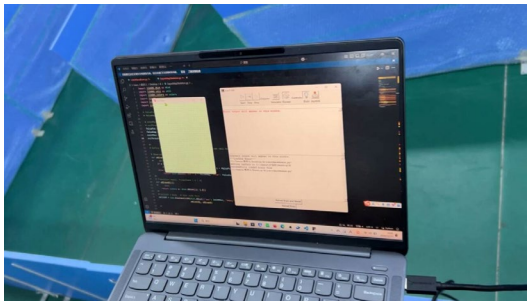




By optimizing the procedure, we were able to increase the robot's running speed and stabilize it until it reached the finish line.

Step 19: A Real Robot

Run on a realrobot, using robotRaceBrain.py. It has a good pair of start-goal values and boundaries for the size of the big world in the front of the room. It will only work on the robot. If you want to test in simulation, you can switch back to using mapAndRaceBrain.py.



In the real world, our robot finally managed to walk out of the maze.

3. Summary

In this experiment, we successfully designed and implemented a dynamic mapping and planning system for a robot navigating through an environment with obstacles. The primary goals included constructing a dynamic map using perfect sonar information, enhancing the map maker to handle imperfect data, and employing Bayesian state estimation for robust mapping. The results demonstrated the robot's ability to dynamically adjust its plan in response to real-time updates from the map maker, even under noisy sensor conditions. However, this noise cannot be too big, and it supports medium noise at most.

Key outcomes include:

1. **Dynamic Mapping**: The robot effectively built and updated its map in real-time, navigating through unknown environments while adapting to changes.
2. **Handling Imperfect Data**: We enhanced the system to incorporate imperfect sensor data, demonstrating resilience through Bayesian estimation techniques.
3. **Integration and Optimization**: The complete system successfully combined mapping, planning, and execution modules, enabling the robot to reach its goal efficiently while avoiding obstacles.

The experiment underscored the importance of combining accurate sensor models, robust algorithms, and real-time planning to tackle challenges in autonomous navigation. Future work may involve improving computational efficiency and enhancing performance under high sensor noise conditions.